Chapter 3. SOAP Data Encoding

When sending data over a network, the data must comply with the underlying transmission protocol, and be formatted in such a way that both the sending and receiving parties understand its meaning. This is what we refer to as *data encoding*. Data encoding encompasses the organization of the data structure, the type of data transferred, and of course the data's value. Just like in Java, it is the data that gets serialized, not the behavior. Data encoding and serialization rules help the parties involved in a SOAP transaction to understand the meaning and content of the message. The model for SOAP encoding is based on XML data encoding, but the encoding constrains or alters those rules to fit the intended purpose of SOAP. I think you'll find that the data requirements of most systems can be easily represented using the encoding rules presented here.

3.1 Schemas and Namespaces

Namespaces provide the mechanism used to determine how element and attribute names are interpreted. Because XML allows arbitrary element names, it needs a mechanism for specifying which dictionary should be used to look up the meaning of any given name. The encoding style defined in section 5 of the SOAP specification is the most commonly used encoding style in SOAP. This encoding style, which is defined in the schema referenced by the `http://schemas.xmlsoap.org/encoding` namespace, is often referred to as "SOAP Section 5." In SOAP messages, this namespace is by convention referenced using a namespace qualifier such as `SOAP-ENC` or `soapenc`. In our examples we will use the `SOAP-ENC` namespace qualifier to refer to this namespace.

It's important to understand that, in SOAP, schemas are used as references to definitions of data elements. They aren't used to validate SOAP message data in standard SOAP processing, although there's nothing stopping you from doing that on your own. References to schemas are often used as namespaces in order to qualify a serialized data element. It's up to the developer or the underlying framework to understand the structure or meaning of the data and to code to it accordingly.

SOAP Section 5 incorporates all of the built-in data types of XML Schema Part 2: Datatypes.[1] This schema defines most of the basic data types you'll use, either directly or as part of your own types. The general practice is to declare a namespace identifier named `xsd` and associate it with the namespace `http://www.w3.org/2001/XMLSchema`. As we saw in the previous chapter, this declaration is usually done as an attribute of the SOAP `Envelope` or SOAP `Body`.

Another common namespace identifier used for data encoding is `xsi`, which is associated with the namespace `http://www.w3.org/2001/XMLSchema/instance`. The easiest way to explain this identifier is with an example. Imagine that we want to declare that the data type of a particular XML element is a `float` as defined by the XML Schema of 2001. We do this by using the `type` attribute from `xsi`, which is an instance data type. The `xsi:type` attribute specifies the data type of the encoded data value. Here's an example of its use:

```
<xyz xsi:type="xsd:float">3.14159</xyz>
```

[1] This spec can be found at http://www.w3.org/TR/xmlschema-2/.

> Note that the schema from 2001 is not the only one out there. Schemas from 1999 and 2000 are also commonly used, and others will emerge. SOAP implementations should be able to handle any of the possibilities because the message will contain a reference to the appropriate schema.

The following namespace identifiers are commonly found in SOAP literature and implementations. Let's take a look at their declarations so that you'll recognize them:

```
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/encoding/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema/instance/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema/"
```

3.1.1 Terminology

In order to fully understand SOAP encoding, it's important to look at some of the terminology. These terms are described in the SOAP spec, but we're going to expand on that a little and look at some examples using both Java and XML.

The term *value* is used to describe the actual data encoded in an XML element. The data is a string of characters. The interpretation of that string can be just about anything you'd normally think of in Java programming: a number, a name, a date, or something more elaborate. The values found in a SOAP message are always associated with a given type. We'll talk about types shortly; for now suffice it to say that the data type of a given value is never undefined in SOAP. There should always be some way to determine the correct type for any value in the message.

SOAP makes a distinction between simple values and compound values. A *simple* value does not contain any named parts; it just contains a single piece of data. The data values associated with programming language types like strings, integers, and floats would all be considered simple values. Here are a few examples in Java with their corresponding examples in SOAP:

```
int a = 10;                     <a xsi:type="xsd:int">10</a>
float x = 3.14159;              <x xsi:type="xsd:float">3.14159</x>
java.lang.String s = "SOAP";    <s xsi:type="xsd:string">SOAP</s>
```

Each of the variables above contains simple values of the associated type. The data is not split into multiple parts, and no other names or mechanisms are needed to get at the data.

A *compound* value contains multiple pieces of data that have some relation to each other. The individual pieces of data may be accessed by indicating an ordinal position in a sequence of values, as with a traditional array. They could be accessed using values that are keys to an associative array, like a hash table. And they could be accessed using the names of the constituent parts, as with a `struct` in C. Whatever the mechanism, there is always a way to distinguish a specific data value within a compound value, and that mechanism is referred to as an *accessor*. It's also possible for a constituent part of a compound value to be a compound value itself. Here are some examples of compound values in both Java and SOAP:

```
int[3] iArray = {10, 20, 30};

<iArray xsi:type=SOAP-ENC:Array SOAP-ENC:arrayType="xsd:int[3]">
    <val>10</val>
    <val>20</val>
    <val>30</val>
</iArray>

class Sample
{
     public int iVal = 10;
     public java.lang.String sVal = "Ten";
}
Sample samp = new Sample(  );

<Sample>
    <iVal xsi:type="xsd:int">10</iVal>
    <sVal xsi:type="xsd:string">Ten</sVal>
</Sample>
```

In this example, ordinal accessors are used to get at the values in the Java array variable `iArray`. To get at the parts of the Java variable `samp`, we use the named accessors `iVal` and `sVal`. We'll look more closely at some compound types, including arrays and structs, in Section 3.3.4.

In SOAP, as in most programming languages, values are associated with an appropriate data type. We've already touched on this in discussing simple and compound data values; each of these values had an associated type. Just like in Java, we declare instances of types and then assign values to them.

3.2 Serialization Rules

The XML elements in a SOAP message are either *independent* or *embedded*. Because of the hierarchical nature of XML, most elements are embedded as subelements of other elements. Independent elements, then, are not subelements of any other elements; they appear at the top level of a serialization.

All of the values in a SOAP message are encoded as the content of an element. Data values cannot appear by themselves outside the confines of an element. That does not mean, however, that every XML element contains a value. For instance, compound data types like structs or arrays contain subelements that contain the actual data values. The elements that define these compound data values do not contain the data directly. Compound types will be covered a little later.

3.2.1 References

SOAP Section 5 also permits elements to reference the values contained in other elements. In this case, no value is provided with the element; instead, an attribute identifies the element in which the actual data value is to be found. The data value must be contained in an independent element, appearing at the top level of a serialization.

The element containing the data value must contain an attribute named `id` of type `ID`. The value of the `id` attribute is the name that other elements use to reference the value. Here is such an element:

```
<lastName id="name-1">Englander</lastName>
```

The `lastName` element is a perfectly valid accessor in its own right. In addition, it has an identifier that allows other elements to reference the data value.

Elements that access a value in another element have no data values themselves. They are empty elements containing an `href` attribute that references the value. The `href` attribute is of type `uri-reference` defined in the XML Schema specification; the value of this attribute references the `id` attribute of the element that contains the data value. Here's an example of an accessor element that references the value of the `lastName` element used previously:

```
<surName href="#name-1"/>
```

This ability to reference the values of other elements is important for a couple of reasons. First, it has the potential to reduce the size of a SOAP message. Imagine that you're sending a message that contains the names of members of the `Englander` family. The XML could look something like this:

```
<member>
    <firstName>Rob</firstName>
    <lastName>Englander</lastName>
</member>
<member>
    <firstName>Jessica</firstName>
    <lastName>Englander</lastName>
</member>
<member>
    <firstName>Arnold</firstName>
    <lastName>Englander</lastName>
</member>
    . . .
    . . .
    . . .
```

As you can see, there's a great deal of duplicated data. References allow us to minimize the repetition because many accessors can refer to the same element. When that occurs, the value is called a *multi-reference* value. Let's take another crack at it using multiple references:

```
<surName id="Last-Name">Englander</surName>
<member>
    <firstName>Rob</firstName>
    <lastName href="#Last-Name"/>
</member>
<member>
    <firstName>Jessica</firstName>
    <lastName href="#Last-Name"/>
</member>
```

```
<member>
    <firstName>Arnold</firstName>
    <lastName href="#Last-Name"/>
</member>
    . . .
    . . .
    . . .
```

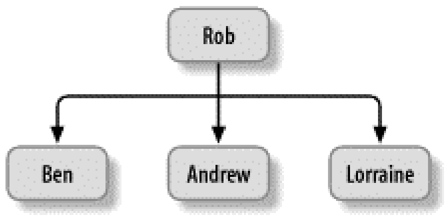
In this case, every member of the family has a `lastName` element that references the value found in the `surName` element. Of course, in this contrived example we haven't really saved much space (if any). However, if the data value were considerably larger, this technique would save considerable space. The savings become more apparent when the value being referenced is a compound element.

Saving space is not the only reason for using multi-reference variables. The technique is also useful when serializing a graph, or collection, of objects where many of those objects have references to the same object. In this case it is important to maintain those relationships when reconstructing the objects during deserialization. Let's look at an example of this in Java. The following code shows a class called `Employee` that contains properties for the employee's first and last names, his or her title, and the employee's manager (also an employee). Following that is some code that defines a manager named `Rob` and three employees named `Ben`, `Andrew`, and `Lorraine`, who all work for `Rob`.

```
Class Employee {
    protected java.lang.String _firstName;
    protected java.lang.String _lastName;
    protected java.lang.String _title = "Worker Bee";
    protected Employee _manager;
    public Employee(java.lang.String first, java.lang.String last,
                    Employee mgr) {
        _firstName = first;
        _lastName = last;
        _manager = mgr;
    }
    public void setTitle(java.lang.String title) {
        _title = title;
    }
    public java.lang.String getTitle(  ) {
        return _title;
    }
    public java.lang.String getFirstName(  ) {
        return _firstName;
    }
    public java.lang.String getLastName(  ) {
        return _lastName;
    }
    public Employee getManager(  ) {
        return _manager;
    }
}
Employee _rob = new Employee("Rob", "Englander", null);
_rob.setTitle("Slave Driver");
Employee _ben = new Employee("Ben", "Jones", _rob);
Employee _andrew = new Employee("Andrew", "Smith", _rob);
Employee _lorraine = new Employee("Lorraine", "White", _rob);
```

Figure 3-1 shows the relationships between the three Worker Bees (Ben, Andrew, and Lorraine), and the Slave Driver manager (Rob). Clearly we wouldn't want to replicate the object referenced by the variable `_rob`. That wouldn't properly represent the relationship between these objects. We want each of the Worker Bee employee objects to reference the same object as their manager. If it's not clear to you why the distinction is important, imagine that `rob` gets promoted to Senior Slave Driver. We wouldn't want to call `getManager( )` on each of the employee objects and then call `setTitle( )`. We would want to make the change once to the `_rob` object.

Figure 3-1. An employee hierarchy



In SOAP, multi-reference variables preserve these kinds of object relationships so that object graphs are represented properly on both sides of a SOAP transaction. The XML for the relationships established in this example could look something like this:

```
<rob id="Rob">
    <firstName>Rob</firstName>
    <lastName>Englander</lastName>
    <title>Slave Driver</title>
</rob>
<ben>
    <firstName>Ben</firstName>
    <lastName>Jones</lastName>
    <title>Worker Bee</title>
    <manager href="#Rob"/>
</ben>
<andrew>
    <firstName>Andrew</firstName>
    <lastName>Smith</lastName>
    <title>Worker Bee</title>
    <manager href="#Rob"/>
</andrew>
<lorraine>
    <firstName>Lorraine</firstName>
    <lastName>White</lastName>
    <title>Worker Bee</title>
    <manager href="#Rob"/>
</lorraine>
```

3.3 Indicating Type

Every element that contains a value must also indicate the type of the data. There are a few different ways to indicate the data type. The first mechanism is to include an `xsi:type` attribute as part of the element. This means that the attribute is named `type`, as defined by the namespace indicated by `xsi`. The value assigned to `xsi:type` must be a valid type identifier such as `xsd:float`, `xsd:string`, etc. In the second mechanism, the value can be an element of an array that already constrains the type of its constituent parts to a particular data type. In this case, no explicit type declaration for the individual values is necessary; we'll see this later when we talk about arrays. Finally, the element name itself can be related to some type that can be determined by looking at the associated XML schema. The following extract from an XML schema defines a compound data type:

```
<element name="Automobile" type="Automobile"/>
<complexType name="Automobile">
     <element name="make" type="xsd:string"/>
     <element name="model" type="xsd:string"/>
     <element name="year" type="xsd:int"/>
</complexType>
```

The data type `Automobile` contains elements named `make` and `model` of type `string`, as well as a `year` of type `int`. Now let's look at an instance of type `Automobile` based on this schema:

```
<niceCar xsi:type="Automobile" >
    <model>Corvette</model>
<make>Chevrolet</make>
    <year>1999</year>
</niceCar>
```

This XML defines an instance of type `Automobile`: a 1999 Chevrolet Corvette. Because the schema for the `Automobile` type already specifies the data types for the constituent parts, we don't need to declare the types in the example.

3.3.1 Simple Types

SOAP makes use of the simple types as defined in the XML Schema Part 2: Datatypes, in the section called "Built-in datatypes." That section of the specification talks about integers, floats, strings, etc., as well as a variety of data types derived from the base types. For instance, a `positiveInteger` is a simple type that is based on the `int` type, but is constrained to allow only positive values.

Just because a data type is declared as a built-in type, don't assume that it will automatically be supported by every SOAP implementation. The smartest thing you can do is check the documentation for the system you're using. You are probably safe with types like `xsd:int`, `xsd:string`, and `xsd:float`. And I think you'll find that quite a few more are implemented pretty much everywhere as well, but just check first.

If you take a look at the XML Schema specification, you'll find a large number of simple types. Table 3-1 shows a few of the simple types and associated example values.

Table 3-1. Simple types defined by XML Schema

Type	Example
int	-41
float	3.14159
string	"Java and SOAP"
positiveInteger	100

Here are the examples from Table 3-1 as they might appear in an XML document:

```
<iVal xsi:type="xsd:int">-41</iVal>
<fVal xsi:type="xsd:float">3.14159</fVal>
<sVal xsi:type="xsd:string">Java and SOAP</sVal>
<pVal xsi:type="xsd:positiveInteger">100</pVal>
```

The data types are explicitly declared using the `xsi:type` attribute so that you can see clearly what the types are. Remember, however, that this is not the only way to associate an element with a type.

3.3.1.1 Strings

You may encounter two different type declarations for string elements. The first is `xsd:string`, which we used in an earlier example. The second is `SOAP-ENC:string`, which is a type based on `xsd:string` that allows the use of the `id` and `href` attributes used for multi-reference values. These string types are not exactly the same as string types in programming languages, such as Java's `java.lang.String`. Some restrictions are placed on the SOAP string types that prohibit the use of special characters, such as those used in forming proper XML markup. Clearly, characters like < and > would interfere with the overall structure of the XML document if they were to appear within a string value. Take a look at this example:

```
<value xsi:type="SOAP-ENC:string">embedded < is no good</value>
```

The inclusion of the < character in the string value is a problem because that character has special meaning in XML. That's not to say that it's impossible to encode a string such as this one; you just have to use replacement characters or some type other than `SOAP-ENC:string`. Escape sequences for the brackets, such as `&lt;` and `&gt;`, are one solution. Another way to resolve this issue is to use MIME base64 encoding, because the base64 alphabet doesn't include any prohibited characters. We'll take a look at base64 encoding shortly.

3.3.1.2 Enumerations

Enumerated values are, essentially, a group of names that represent other values. In the C programming language, the `enum` keyword allows you to use descriptive names in place of integer values. For instance, instead of using the values 1, 2, and 3 to represent the colors red, yellow, and green, you might define an enumerated type that uses the color names in place of the values. At runtime, the system replaces the names with the associated values. This is more or less a programming convenience that can lead to more readable and understandable source code.

One problem with enumerated types is that the values used to represent them internally are not standardized. There is no standard integer value that represents the name `Green`, for instance. So there is an inherent problem with transmitting enumerated values from one system to the other. The actual values used internally are more than likely unique to the application. In the C language, the values often default to an integer sequence starting with 0. In this case, the order in which the names are defined in the enumeration type determines the values; the first entry is defined as 0, the second as 1, and so on. However, the syntax allows the programmer to override that behavior and define each entry independently. So the actual values are a programming detail not likely to be the same from one program to the next. Some programming languages, like Java, don't even support enumerations.[2] But that doesn't diminish the usefulness of enumerated types.

SOAP encodes enumerated types by encoding the name of the value. In the case of the traffic signal, the SOAP encoding ignores the internal implementation value in favor of the name of the color. Here is an XML snippet that shows the encoding of the state of an imaginary traffic signal:

```
<trafficSignal>
    <signalState xsi:type="xsd:string">Green</signalState>
</trafficSignal>
```

The sending application does not encode the internal representation of the value of `Green`; instead it uses the name `Green` itself. It's up to the recipient to resolve the value `Green` into its own appropriate internal value. It's also possible to define an enumeration type in a schema. Here's what the traffic signal schema definition might look like:

```
<element name="TrafficSignalState" type="TrafficSignalState"/>
<simpleType name="TrafficSignalState" base="xsd:string">
     <enumeration value="Red"/>
     <enumeration value="Yellow"/>
     <enumeration value="Green"/>
</simpleType>
```

Using this schema definition, we could encode the `trafficSignal` value as:

```
<trafficSignal>
    <TrafficSignalState>Green</TrafficSignalState>
</trafficSignal>
```

3.3.2 Binary Encoding

SOAP recommends that binary data, referred to in the SOAP specification as *byte arrays* , be encoded using the MIME base64 encoding algorithm. Since there is no data that cannot be represented in binary, this means that a SOAP document can encode any data type using the base64 character set. This has the added benefit of substituting characters that may have special meaning to certain protocols.

Although it's not appropriate to cover all the details of base64 encoding here, it might be useful to walk through a simple example. Base64 encoding uses an alphabet composed of 6-bit values. Each of these values has a defined substitution character. Table 3-2 shows the base64 alphabet.

[2] Java, of course, has an `Enumeration` interface, which is a mechanism for accessing the elements of a collection. That feature isn't related to a C-style enumerated type.

Table 3-2. Base64 character set

6-bit value	Base64 character
0-25	A - Z
26-51	a - z
52-61	0 - 9
62	+
63	/
PAD	=

To encode your binary data you must break it down into 6-bit pieces. If the total number of bytes in your data is not divisible by 6, fill it out to the right with zeros. Base64 encoding was designed for email, which constrains each line of data to 76 characters. In the SOAP world we are not bound by that limit, so you don't have to break the data on that boundary; you can insert new lines anywhere convenient.

Let's say we want to encode the string "<data>". Those braces are a problem in string encoding, but it'll work out fine in base64. First, let's look at the hexadecimal representation of each byte of data:

Character	<	d	a	t	a	>
Hex value	0x3C	0x64	0x61	0x74	0x61	0x3E

Next we'll convert to binary and break it up into 6-bit pieces. This gives us exactly 8 pieces, so we don't need to pad the data. Here are the 6-bit binary pieces along with their decimal equivalents and associated base64 characters:

6-bit value	001111	000110	010001	100001	011101	000110	000100	111110
Decimal value	15	6	17	33	29	6	4	62
Base64 value	O	F	P	h	d	F	D	+

Now we can encode the data using base64 encoding and represent it in XML as:

```
<value xsi:type="SOAP-ENC:base64">OFPhdFD+</value>
```

To do the decoding, first convert each character of the string into its equivalent 6-bit value. Next regroup the bits into 8-bit values, using each one to form one byte of the decoded data. You should get back to the original string: "<data>".

Base64 encoding can be useful for transmitting any kind of binary data. However, if you're thinking about using this encoding style to transmit a 10 MB bitmap file, you may want to think again. Base64 increases the size of the data by about 1/3, which can be significant in some environments. That's not to say it won't work, of course, and it may be just fine for your application.

One common technique for handling large data objects is to send a URL that references the data instead of sending the data itself. The recipient can then use that URL to retrieve the data out-of-band. This is a good technique when you want to keep your SOAP message small, especially if the data may not be used by the recipient.

Another technique, which we'll talk about in Chapter 8, is to send the data as a MIME attachment. This doesn't reduce the overall size of the message, but it does keep the data out of the SOAP payload, improving the performance of the SOAP processing.

3.3.3 Polymorphic Accessors

The term *polymorphism* is used frequently in object-oriented software design and development. It means that a thing (usually an object) can take on the appearance of other things. Although this book is not about object-oriented programming, let's talk about what we usually think of as polymorphism in Java and then look at what it means in SOAP.

Consider our earlier Java class called `Employee`. We used the `Employee` class for managers as well as regular employees. Of course in the real world there are characteristics and behaviors of managers that are not present in regular employees (no, I don't mean personality traits, although we're all aware of those differences!). For instance, a manager class may include a collection of employees under his or her care (or control, depending on your perspective). We could create a `Manager` class that derives from the `Employee` class, and then add the manager-specific stuff to that new class. This works because managers are also employees. In other words, managers are a type of employee, so an instance of a `Manager` object can also take the shape of an `Employee` object. If the `Manager` instance is passed to some function as an `Employee` reference, the receiving method will not know or care that the object is actually a `Manager` instance. It sees it as an `Employee` instance. This is an example of polymorphism.

Another example of polymorphism is the variant type in Microsoft COM (Component Object Model) programming. The variant represents a data value and an associated data type indicator. The data type indicator tells the user which accessor to use when retrieving the data.

In SOAP, the mechanisms of polymorphism are not nearly as elaborate as they are in Java; SOAP polymorphism more closely resembles the COM variant. If you've looked at the SOAP specification, you may have noticed that the entire section on polymorphism is just a few lines long. Polymorphism in SOAP means only that an accessor includes a type declaration. That's it. The following XML fragment shows an accessor named `quantity`. In the first case, assume that `quantity` is defined to be of type `xsd:int` according to some schema. Therefore, no type declaration is used. The value cannot take on different types. In the second case, the `quantity` accessor includes a type declaration, allowing the accessor to take on a data type of `xsd:float`. The second form is a polymorphic accessor.

```
<quantity>37</quantity>
<quantity xsi:type="xsd:float">37</quantity>
```

We'll see some important uses of polymorphic accessors when we look at heterogeneous arrays and generic compound types.

3.3.4 Compound Types

The SOAP specification talks mostly about two kinds of compound data types: structs and arrays. Here's how the spec defines these types:

Struct

> A compound value in which accessor name is the only distinction among member values, and no two accessors have the same name.

Array

> A compound value in which ordinal position is the only distinction among member values.

We touched on these definitions earlier when we talked about the terminology of SOAP encoding. These data structures are common to many programming languages, and there's nothing different about the way SOAP defines them.

3.3.4.1 Structs

Many programming languages support structs, though not all in the same way. For instance, in the C programming language you can define a compound structure using the `struct` keyword. This keyword essentially allows you to define a data type or instance that comprises named parts, where each part can be an instance of any other valid data type. In C++ you can use the `struct` keyword in the same way you can in C. However, you'd most often use a special (and more functional) structure called a class. In Java there is no explicit concept called a struct, but just like in C++ we can use a class to provide the same thing as a struct.

Here is a Java class that contains fields representing some characteristics of a guitar:

```
class Guitar {
    public java.lang.String _manufacturer;
    public java.lang.String _model;
    public int _year;
}
```

This may not be the best way to represent a guitar in a real-world Java application, but it is perfectly valid and closely resembles the `struct` available in C and C++. An XML schema fragment corresponding to this class could look like the following.

```
<element name="Guitar" type="Guitar"/>
<complexType name="Guitar">
     <element name="manufacturer" type="xsd:string"/>
     <element name="model" type="xsd:string"/>
     <element name="year" type="xsd:int"/>
</complexType>
```

Here's an example of a serialized `Guitar` instance using this schema:

```
<Guitar>
    <manufacturer>Epiphone</manufacturer>
    <model>Sheraton II</model>
    <year>1997</year>
</Guitar>
```

3.3.4.2 Arrays

Instances of SOAP arrays are declared using the data type `SOAP-ENC:Array`. The elements of an array can be any valid SOAP type, even arrays themselves. Arrays must have a `SOAP-ENC:arrayType` attribute that specifies the data type of the elements as well as the dimensions of the array.

Let's start off by looking at an array declaration for a one-dimensional array of five floating-point values. The actual values are omitted for now (we'll get to that subject next).

```
<someNumbers xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:float[5]">
    . . .
    . . .
</someNumbers>
```

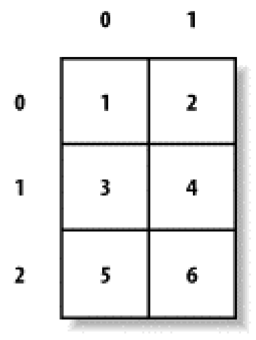
An array is serialized as a series of element values, where the order of the elements represents their ordinal positions. The actual element name for each value in the array is not significant because array members are accessed by position; XML requires that some name be used for each array element, but SOAP doesn't care what the name is. Often you'll find that every element of a serialized array uses the same name, but that's not a requirement. The SOAP specification suggests using the element's data type as the element name. That, too, is only a suggestion. I like to use names that reflect the contents of the array if at all possible. Here is an array with the data values filled in:

```
<someNumbers xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:float[5]">
    <number>1.234</number>
    <number>2.345</number>
    <number>3.456</number>
    <number>4.567</number>
    <number>5.678</number>
</someNumbers>
```

Since the elements are serialized in the order they appear in the array, you can determine their ordinal position by inspection. Like Java and many other languages, SOAP starts array indexing at 0. So the value 4.567 is at ordinal position 3 (the 4th element of the array).

In the case of multi-dimensional arrays, elements are listed in row-major order. This means that the ordinal position farthest to the right changes the fastest. Figure 3-2 is a two-dimensional array containing the integer values 1 through 6. The dimensions of the array are 3 by 2, and the figure shows the major dimension down and the minor dimension across.

Figure 3-2. A two-dimensional array



The `SOAP-ENC:arrayType` allows you to specify the size of each dimension of a multi-dimensional array by separating each dimension size with a comma. Here's what the encoding of the array in Figure 3-2 might look like:

```
<values xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:int[3,2]">
    <value>1</value>
    <value>2</value>
    <value>3</value>
    <value>4</value>
    <value>5</value>
    <value>6</value>
</values>
```

So far, the arrays we've looked at are homogeneous: every element of the array is of the same data type. But it's also possible to encode heterogeneous arrays, where each value is of a different type. This is one place where polymorphic accessors are important. To create a heterogeneous array, the `SOAP-ENC:arrayType` is assigned the value `xsd:ur-type`. This type means that the data types of the enclosed array values are not being constrained by the `SOAP-ENC:arrayType` attribute, but instead each data value is encoded to include its data type. Here is an array that contains values of different types:

```
<mixedNuts xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:ur-type[3]">
    <nut xsi:type="xsd:float">3.14159</nut>
    <nut xsi:type="xsd:string">SOAP</nut>
    <nut xsi:type="xsd:int">8141992</nut>
</mixedNuts>
```

SOAP also permits the size of an array dimension to be omitted. This means that the number of values in a particular dimension of the array can be determined by looking at the members that are encoded in the array. So the first line of the `mixedNuts` example could also be written as follows:

```
<mixedNuts xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:ur-type[]">
```

By inspecting this XML, we are able to determine that this one-dimensional array is of size 3.

3.3.4.3 Partial arrays

In some cases an array won't be fully populated. In other words, there may be one or more elements in an array that have no value. In the following Java code, the array `contestants` is a one-dimensional string array of size 10, but only the elements at positions 6, 7, and 8 contain data values:

```
java.lang.String contestants[] = new java.lang.String[10];
contestants[6] = "Rob";
contestants[7] = "Arnold";
contestants[8] = "Scott";
```

If we were to use the standard encoding for this array, we'd need to encode every element, even though most of those values are null. Null values shouldn't be confused with empty values. In the case of our string array, there is a difference between an element that contains an empty string and an element that is null. We certainly want to encode empty strings because that properly reflects the data that exists. In the example above, the values in the `contestants` array at all positions other than 6, 7, and 8 are null; they do not contain instances of `java.lang.String`. We could encode those null elements by supplying an `xsi:null` attribute for each one, with a value of 1. These elements won't contain any data:

```
<contestants xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:string[10]">
    <name xsi:null="1"/>
    <name xsi:null="1"/>
    <name xsi:null="1"/>
    <name xsi:null="1"/>
    <name xsi:null="1"/>
    <name xsi:null="1"/>
    <name>Rob</name>
    <name>Arnold</name>
    <name>Scott</name>
    <name xsi:null="1"/>
</contestants>
```

This is perfectly fine, but it's not very efficient. SOAP allows you to encode this kind of array by omitting everything but the block of elements that contain values. To do so, include the `SOAP-ENC:offset` attribute in the array declaration. The value of this attribute represents the ordinal (zero-based) position of the first element actually included in the array. So we could encode the `contestants` array as follows:

```
<contestants xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:string[10]"
             SOAP-ENC:offset="[6]">
    <name>Rob</name>
    <name>Arnold</name>
    <name>Scott</name>
</contestants>
```

This is certainly more efficient than transmitting all the null array values. Note that the `SOAP-ENC:arrayType` attribute still specifies that the array has a size of 10. This is important, as we want the receiver of this array to use the proper array size.

3.3.4.4 Sparse arrays

The use of the `SOAP-ENC:offset` attribute works perfectly if the data values present in the partially transmitted array are contiguous, as they were in the `contestants` example. But certainly this isn't the only possible scenario of partially populated arrays. What if the data values are in noncontiguous positions, as in the following code?

```
java.lang.String contestants[] = new java.lang.String[10];
contestants[1] = "Rob";
contestants[5] = "Arnold";
contestants[8] = "Scott";
```

This is what's known as a sparse array. SOAP provides for sparse arrays by allowing each transmitted element of the array to specify its own ordinal position by including a `SOAP-ENC:position` attribute. So we can now encode the array as:

```
<contestants xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:string[10]">
    <name SOAP-ENC:position="[1]">Rob</name>
    <name SOAP-ENC:position="[5]">Arnold</name>
    <name SOAP-ENC:position="[8]">Scott</name>
</contestants>
```

The encoding of partially transmitted and sparse arrays has the potential of being far more efficient than encoding all of the null values in a partially populated array. You should be aware, however, that many of the current SOAP implementations don't support these constructs. I'm sure that over time, and hopefully by the time you read this, SOAP implementations will support partial and sparse arrays.

3.3.5 Generic Compound Types

There are times when the exact format of a serialized compound data value is not known in advance. This might be because there is no schema definition for the data type, or it might occur as the result of a general database query. For example, let's say you are using SOAP to query a database by sending an SQL query like the one shown here:

```
select * from guitars;
```

In case you're not familiar with SQL syntax, this statement asks for all the records from the table named `guitars`, and returns all the fields from each record in that table. However, we may not know in advance the names of the fields, or even how many fields there are. Table 3-3 represents a simple database table called `guitars`, containing two records. The table contains three fields: `manufacturer`, `model`, and `year`.

Table 3-3. The guitars table

Manufacturer	**Model**	**Year**
Gibson	Les Paul	1958
Fender	American Stratocaster	2001

Even though we don't know in advance the fields to expect back from our SQL query, a valid SOAP response could contain the following serialized data:

```
<guitars>
    <guitar>
        <manufacturer xsi:type="xsd:string>Gibson</manufacturer>
        <model xsi:type="xsd:string">Les Paul</model>
        <year xsi:type="xsd:int">1958</year>
    </guitar>
    <guitar>
        <manufacturer xsi:type="xsd:string>Fender</manufacturer>
        <model xsi:type="xsd:string">American Stratocaster</model>
        <year xsi:type="xsd:int">2001</year>
    </guitar>
</guitars>
```

This XML contains generic compound data. It's up to the receiver of this data to figure out how to deal with it. Both struct and array constructs can be used within generic types. The format of each `guitar` element resembles that of a SOAP struct, and the `guitars` element looks very much like an array, even though it isn't declared using `xsi:type="SOAP-ENC:Array"`. The elements of each guitar use polymorphic data accessors, since their data types may not be known in advance.

3.4 Default Values

The SOAP specification talks briefly about the handling of default values. It states that the omission of an element from the transmitted data can imply that a default value is to be used. The C++ language provides a good programming analogy to this topic, since it allows you to define default values for method parameters, thus allowing the caller to omit values for those parameters. Here's what just such a method definition in C++ looks like:

```
int getValue (int param1, int param2 = 1);
```

In this case, the default value for `param2` is 1. So if the caller doesn't provide a value for that parameter when calling `getValue( )`, the system uses the value 1. Although there is no support for this kind of syntax in Java, it is certainly possible to define SOAP transactions that support default values.

Let's say that an order entry system expects a compound element named `order`, where each item in the `order` contains a quantity, an item number, and a description. We might want to say that the default value for the quantity is 1, so that the sender is not required to include a quantity for an item if they are ordering only one. In this case we might serialize an order as:

```
<order>
    <item>
        <quantity>2</quantity>
        <itemNum>1020304</itemNum>
        <description>Really cool device</description>
    </item>
    <item>
        <itemNum>1020305</itemNum>
        <description>Some other device</description>
    </item>
</order>
```

We did not include a quantity element for the second item. Instead, we expect the recipient to use the default value. The important point is that SOAP does not define default values, nor does it mandate them. It's up to the sender and recipient to do the right thing.

3.5 The SOAP Root Attribute

Linked lists, trees, and directed graphs are common data structures. However, we've yet to see a serialization mechanism that identifies where the starting point, or root, of such a structure might be. The SOAP root attribute is used for just this purpose. Assigning the value of 1 to the `SOAP-ENC:root` attribute of an element identifies it as the root of a structure such as a linked list.

Here is some Java code for the nodes of a singly linked list of integers, followed by an instance of a list containing three nodes. The nodes contain the values 50, 60, and 70, and they are placed into the list in ascending order with the node containing the value 50 being the head (or root) of the list.

```
class Node {
    int _value;
    Node _next;
    public Node(int value, Node next) {
        _value = value;
        _next = next;
    }
}
Node n3 = new Node(70, null);
Node n2 = new Node(60, n3);
Node n1 = new Node(50, n2);
```

The variable `n1` is the root of the list. Here's what the serialization for this list might look like:

```
<list>
    <node>
        <value id="node-3" xsi:type="xsd:int">70</value>
        <next xsi:null="1"/>
    </node>
    <node>
        <value id="node-2" xsi:type="xsd:int">60</value>
        <next href="#node-3"/>
    </node>
    <node>
        <value SOAP-ENC:root="1" id="node-1" xsi:type="xsd:int">50</value>
        <next href="#node-2"/>
    </node>
</list>
```

We used value references to link one node to the next, and the `SOAP-ENC:root` attribute to indicate the head of the list.